Chapter 1 - Introduction

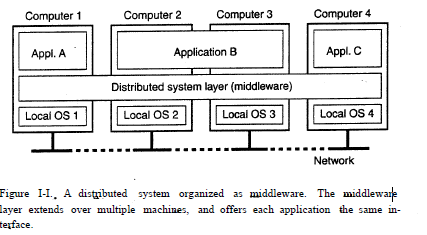
Technology was expensive, but the advent of LANs and WANs changed that. LANs are a collection of connected computers in one building and the way they are connected allows data to flow very fast between machines. A WAN is the same idea, but instead of the machines residing in one building, they reside could all over the globe.

Distributed System – network of independent machines that appear to the outside world as a single unit

Independent – autonomous, machines do not have a dependence on each

Single unit – users see it as a single entity and are not aware of the intricacies of communication between machines, they don’t know when a machine has be swapped out or removed altogether

To allow a distributed system to be seen as independent and autonomous, it usually resides between the user/application layer and OS layer. Since it is sandwiched between two layers, it has come to be known as middleware. Here is Figure 1.1 from Distributed Systems: Principles and Design, Tanenbaum and van Steen, Second Edition



Aims of a Distributed System

Easily accessible remote resources

This makes good economic sense. Rather than have a printer for each employee, it is cheaper to have one printer and have employees in the general vicinity of the printer to share it. Having employees be able to use remote resources also makes collaboration with other employees is easier and this has become very popular. So popular that there is now software, collectively called groupware, that makes it possible for employees to share files, video conference meetings, etc. This sharing comes at a price in that security becomes a very real concern as passwords and other sensitive information are sent unencrypted over the network.

Transparency

Here is Figure 1.2 from Distributed Systems: Principles and Design, Tanenbaum and van Steen, Second Edition, with an “Example” column added.

|  |  |  |
| --- | --- | --- |
| Transparency | Description | Example |
| Access | Hide differences in data representation and how resource is accessed | File naming schemes could differ on computers; int could be 32 bit on one machine and 64 bit on another |
| Location | Hide where resource is located | User has no idea where in the world a server he/she is connected to located |
| Migration | Hide that a resource may be moved to another location | User has no idea the server they are connected to was always at that location or was just moved there recently |
| Relocation | Hide that a resource may be moved to another location while in use | Laptop user in a train does not lose a wi-fi signal |
| Replication | Hide that a resource is replicated | User doesn’t know that there are clones of the resource; To support this, the system should also support location transparency |
| Concurrency | Hide that a resource may be shared by several competitive users | When accessing a table in a database, a user sees consistent view of data and is not aware that other users are updating the same table |
| Failure | Hide that failure and recovery of a resource | User doesn’t know that resource crashed and is back up; Not always a good thing, you may want to let people know that a failure occurred |

There is also the issue of degree of transparency, i.e. how much to hide from the user. In some cases, the user would need to know what is going on in the distributed system. For example, if processes across the world need to connect, the user needs to aware of the lag since it will be noticeable. Another example is updating all clones of a resource, say a file. This will cause updates to take noticeably more time and the user should be notified of this fact. Transparency is a lofty goal but full transparency is impossible.

Openness

Openness in a distributed system means there is a well-defined interface to access the services offered by the distributed system. These interfaces are specified using Interface Definition Language (IDL) and they should be complete (i.e. the interface contains all that is needed to create an implementation) and generic (i.e. they do not give any indication as to the actual implantation). Complete and generic interfaces allow for interoperability (multiple implementation of the same interface can communicate and get a task completed), and portability (applications implemented for one distributed system can run as is on another without incident). You also want your distributed system to be able to easily add more functionality.

Newer distributed systems have moved away from having only interfaces defined for the highest level interfaces and have added interfaces to specify the interaction of lower level components as well. This allows for more ease when adding/modifying functionality.

3 Types of Scalability Issues

Size

Increasing the size of a distributed system (i.e. adding more computers and users) brings about its own set of issues when you have centralized services, data, or algorithms. They can become overloaded and impact performance when taxed very heavily. It is easy to see that having one server handle the requests of every user in the system could cause problems once the number of users becomes large. Similarly, having all the data in one place can pose problems when there are a lot of users accessing this data. In both instances, users will experience a noticeable delay in the response to their requests to a server and in the case of the lone database, users will notice delays when accessing and/or updating tables. Lastly, decentralized algorithms have the following desirable characteristics, Distributed Systems: Principles and Paradigms, Tanenbaum and Van Steen, Second Edition

1. No single machine has all of the information on all machines
2. Machines make decisions based on local information
3. Failure of one machine does cause algorithm to crash
4. No global clock (not possible for machine to know time on another machine)

Geographic

Challenge here is for distributed systems for LANs, where the client waits for a response from a server. This is fine for LANs where the machines are in the same building, but when a request is sent from a client to a server on opposite side of the globe, the client will experience a significant delay. The answer to this is to have asynchronous calls, wherein the client makes the request and instead of waiting, the client continues doing other things. However, this is more complicated to implement. LANs allow for reliable data transfer and support broadcasting. However, WANs offer no guarantees and broadcasting to all the machines on a WAN is not realistic.

Administrative Domains

A distributed system inside one domain has tested applications and the system administrators have locked it down pretty well so that it cannot be attacked. If this distributed system then crosses into another domain, it needs to have security measures in place to defend itself against any attacks from the new domain.

3 Scaling Techniques

***Hiding Communication Lags***

This is applicable for solving issue with geographic scaling. Namely, the client should use asynchronous calls. However, this will not work when asking for user input (i.e. filling out a form) and validation is needed. The client will enter the information and wait until the server validates the information in the form’s fields. In this case, it is better to handle validation at the client end, and once it is correct, send the form to the server.

***Distribution***

This entails breaking components down and scattering them across the system. DNS is an example of this. The naming service of DNS sends parts of the name to machines at various levels in a hierarchy in order to get the IP address

***Replication***

This involves having copies of resources available throughout the distributed system. This is useful when the distributed system spans the globe and the distance between machines is very great. If copies are placed closed to users, then the users will not experience delays. However, this leads to the possibility of inconsistent copies of the data in the system.

Distributed Computing Systems - Cluster Computing

Instead of spending money on a very fast computer with a lot of memory, to perform a data intensive task, the trend has been to use many cheaper (highly similar) machines with high speed connections. An example of this is Hadoop.

Distributed Computing Systems - Grid Computing

The machines used in grid computing are very different wrt to OS, hardware, etc. Software for grid computing is has a 4 layer architecture:

*Fabric layer*

*Connectivity and Resource layer*

*Collective layer*

*Applications layer*

Distributed Information Systems

Applications are large and complex so there was a need to break them down and allow them to talk to one another. This led to focusing on EAI (Enterprise Application Integration).

Transaction Processing Systems

4 characteristics of transactions (ACID):

Atomic

To users, transaction happens indivisibly; all parts of transaction are executed or none at all.

Consistent

The transaction does not violate invariants. During the transaction, invariants can (and will) be violated but once the transaction is complete, the invariants will hold.

Isolated

Concurrent transactions do not interfere with each other.

Durable

Once a transaction commits, changes are permanent and cannot be undone. Only applies to top-level transactions

Instead of a single database, transactions can span multiple databases. For example a transaction (A) can spawn transactions (B and C) in 2 databases, db1 and d2, respectively. Transaction A is called the top-level transaction and B and C are each a subtransaction. This type of transaction is called a nested transaction. As stated above, the durability only applies to the top-level transaction. That is, if the top-level transaction, A, aborts after subtransaction B commits, the commit by B must be undone.

Enterprise Application Integration

Applications can communicate with one another via RPC (remote procedure calls). This allows an application to call a service in another application on another machine. RMI (remote method invocation) allows objects in different application on different machines to interact with one another. (Read up on RMI ???). For RPC and RMI, both applications need to be up. This disadvantage is addressed using message-oriented middleware (MOM), i.e. a publisher/subscriber model.

Distributed Pervasive Systems

In a distributed pervasive system, the components are everywhere i.e, IoT (Internet of Things)

Examples:

Home system – laptops, watches, appliances, etc.

Electronic Health Care systems – worn on the body, the devices monitor biological signs and inform doctor when needed

Chapter 2 – Architectures

Two types of system architectures are discussed:

Centralized – one server in the distributed system provides services and clients send requests to this server for these services

Decentralized. – all machines in the distributed systems act as servers and clients. That is, all machines make requests respond to requests. Example: BitTorrent

Autonomic distributed system – a distributed system that is self-managed. It can detect issues and respond to them

Component – in a distributed system, it is a self-contained unit of software that can be swapped out as needed, provided that its interfaces are adhered to

Connector – means by which components communicate and collaborate, i.e. RPC/RMI

3 Architectural Styles

Layered – components are in layers and components at layer N can call services in layer N-1, but not vice versa. This is like the OSI model.

Object-based – components are modeled as objects and communication between them is achieved via RMI

Data-centered – communication between processes is via a single data store

Event-based – Publisher /subscriber model; one process publishes (broadcasts) events/data and other processes subscribe to receive those events/data. Middleware has the job of making sure that only those that subscribe to a certain event/data, get that event/data. The 2 processes are independent and don’t need to know about each other’s internals at all. Can combine with data-centered architecture wherein the shared data resides in the cloud.

2.2 – System Architectures

Software architecture defines how software components are structured and how they communicate with one another. System architecture defines the structure and behavior of a system (collection) of machines, with these software components running on them.

Centralized Architectures

A server provides a service to clients that make requests to it for this service. Client and server can communicate via UDP but there is no guarantee that data will not be lost. This would be problem when you are sending requests to debit an account. If the message fails and is sent again then the account may be debited twice. However, idempotent requests are fine since they are read only. For example, repeatedly requesting an account balance would not cause problems. Some systems use TCP/IP instead of UDP because message delivery is guaranteed without loss of data, but this is an expensive solution.

Application Layering

In order to delineate client/server roles, a system is divided into layers (levels) by an application’s role:

User-interface level

Allows applications to interact with the user, i.e. getting input from the user, and provides a GUI (simple or fancy) to the user to facilitate such input

Processing level

Contains the crux of the functionality of a system. For a search engine, this corresponds to taking the input string from a user and making a database query (or queries) out of it, ranks the result of the query (or queries), and make an HTML page out of the results. Financial software can have a GUI in the front end and a database of financial data at the back. The processing level would be between the two, taking input from the front end and crunching data from the database (i.e. running financial models on the data), and giving results to the user.

Data level

Holds and manages the data used by the applications in the system. The data must be persistent, that is, it must be available from one session to the next. This is usually a relational database, e.g. a SQL database. The database can have triggers to notify administrators of events of note, namely, a balance going negative. Object-oriented databases are also used, for example, when the business domain consists for complex objects that do not lend themselves easily to the relational database paradigm.

Multi-tiered Architectures

In the client server model you can have:

One machine that acts as a client (making requests)

and

Another that acts as a server (fulfilling requests).

There are other ways in which the work may be divided. That is, put more ‘smarts’, i.e. functionality into the front end. That is, the client and server can have varying responsibilities for the user, application and database levels. That is, the client can be responsible for the user interaction level, and also have some functionality related to the processing level i.e., contain some minimal business logic. For example, a client can handle displaying a form to the user and validating the form (per some business rules) before sending it on the server. This can go the extreme end where the client machine is responsible for the front end as well as the business logic and all the server end does (for example) is perform database lookups.

Thin vs fat clients – thin clients are end user machines contain very little business logic and provide only an interface to the user, whereas fat clients are end user machines that perform quite a bit of the business processing. Fat clients are not preferable because having all the processing done on client machines would be very difficult to manage. It is much easier to have a thin client with very little business logic and put the processing/storing of data in a centralized location.

In modern systems, the servers are not always on one machine, but are running on many different machines. In this situation, the server can in turn act as a client. For example, when the client makes a request of a server for information in a database, and the server in turn makes a request to another server to get the data.

Decentralized Architectures

Vertical distribution – in a vertically distributed system, the user-interface level, the processing level, and the data level are scattered across different machines.

Horizontal distribution – in a horizontally distributed system, each level is distributed across all machines in the system so that all machines can act as clients and servers, (and are called servents), thus lessening the load on any given machine. Example: peer to peer systems

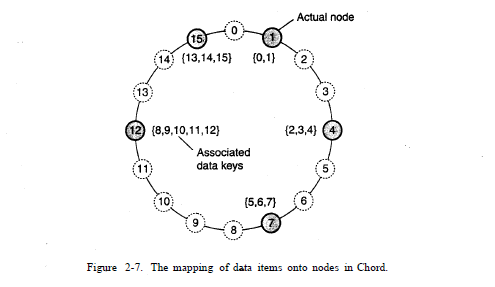
Because the machines in peer to peer systems can act as clients and servers, their architecture develop by addressing the problem of ordering the processes in an overlay network, i.e., a network on top of another network where the nodes are the processes and the links between nodes are communication channels that are used by the processes to communicate since the processes cannot talk directly to each other.

2 Types of Overlay Networks – structured and unstructured peer-to-peer architectures

Structured Peer to Peer Architectures

Chord

Overlay networks are created deterministically. A popular method for this is to use a distributed hash table (DHT). Data are assigned a random key and nodes are assigned random ids. At the heart of the DHT system is to map keys to ids. The Chord system is such an example, and referring to the Figure 2-7, works as follows. Nodes are arranged in a ring and the data items are assigned keys and mapped to a node such that the node id is greater than or equal to the keys of the data items for that node. For example, in the figure, node 12 has data items with keys less than or equal to 12, i.e. the data item keys are 8, 9, 10, 11, and 12. Also, node 12 is called the successor of key k, i.e. succ(k). To find a data item with key k, an application calls LOOKUP(k), which returns the address of succ(k). Now, the application can contact the node to get a copy of the data item.



Membership management (Chord)

Membership management refers to how nodes enter/leave the system. In Chord, a node wanted to enter a system generates an id, does a LOOKUP(id) to get the address of succ(id) and and contacts the node (succ(id)) at that address, and its predecessor and insert itself into a ring. When leaving, the node notifies its predecessor and successor of its departure and gives it data items to succ(id).

Content Addressible Network (CAN)

CAN is another DHT system. Here, n-dimensional Cartesian space is divided into regions and nodes are assigned to the divided regions (one node covers one region) and data items are also assigned to points in this partitioned Cartesian space. This means that the data items that fall within a certain region are associated with the node for that region. To join, a node picks a point and finds the region in space that corresponds to this point. The node associated with this region then splits the region in half, and one half of the region is associated with the new node that is joining.

Leaving is more difficult than in Chord, and could mean asymmetric partitioning of the Cartesian space.

Unstructured Peer to Peer Architectures - ???

Overlay networks are created using randomized algorithms.

Each node has list of neighbors that are current live nodes. This list is known as a partial view.

Topology Management of Overlay Networks - ???

2 layers:

Lower layer is an unstructured peer to peer system where nodes aim to maintain a random graph that consists of live nodes. This partial view is then passed to the higher layer

Higher layer – more neighbors are selected and nodes may be ordered per a ranking function

Superpeers

Finding data items in unstructured peer to peer architectures is difficult since their overlay networks use randomized algorithms. This means that to find an item, a node must blast the network with a query. Because of this, there are times when you do not want to have machines that can act as clients and servers at the same time. Instead, it is more sense to have nodes that are superpeers. These are nodes that act as a server for some nodes and contain indexes into the data that would be of interest to that subset of nodes. When a peer joins a network, it is associated with a superpeer and this association remains until it leaves the network. This is not the best plan since it is possible that a peer may be associated with a superpeer that doesn’t have the data items it needs.

Hybrid Architectures - Edge Server Systems

An edge server is on edge of two networks. For example, in the case of a home network and the Internet, the ISP is an edge server.

Hybrid Architectures – Collaborative Distributed Systems

BitTorrent

BitTorrent is a file sharing system that uses a peer to peer architecture. A user must download a BitTorrent client and then go to web site to access a directory that has references to .torrent files which contain ‘trackers’. These trackers are servers that keep track of the computers that are connected and give their IPs to other clients which allow them to connect to one another. Once connected, the chunks of the file are downloaded from active users. Since this is a file sharing system, you are expected to be a good citizen and upload as much (or more) than you download. If you do not, others will not share as much with you.

Globule - ???

Network comprised of Web servers that can host the pages of the websites of other users.

Chapter 3 - Processes

3.1 - Threads

A process is a running instance of an application (lot of code) whereas a thread runs code within a process (much less code). Threads share memory whereas processes do not. Both processes and threads run independently of one another but since threads share a memory space special care must be taken when threads manipulate the same data. An application can have several threads running within it at the same time. This means that when an application contains threads (i.e. a multi-threaded application), the developer needs to write the code properly so that threads do not interfere with one another in an inappropriate way, causing the application to behave strangely. For example, if 2 threads are accessing the same variable. If the code is not written properly, it is possible that Thread A will update it and before the update is complete, Thread B goes to read from it. Since the update has not completed, Thread B will be looking at an old value and could make wrong decisions based on this. Processes do not share memory so this is not a concern for processes. Writing multi-threaded code is challenging but pays dividends when it comes to performance.

Threads give the appearance that tasks are being performed simultaneously. However, when there is one CPU, tasks are just being (very quickly) switched from on thread to another or one process to another to give the user the appearance that things are being done all at once.

The processes run by an OS are maintained in a process table. The process table contains information about the processes the OS is running, i.e. a snapshot of each process. The CPU is running multiple processes at the same time so it needs to switch between processes (i.e. context switching). For example, if a CPU is running process A and needs to switch to process B, it saves a snapshot of A to the table, picks up B’s snapshot from the table and runs B for bit, and when it is time to run A again, it puts B’s snapshot back and gets A’s snapshot and continues running A. Saving/accessing all of this information makes context switching expensive.

In contrast, since the OS does not supply each thread with its own space and there is no promise by the OS that threads can safely concurrently, there is very little information to maintain.

Thread Usage in Nondistributed Systems

Multi-threaded applications leverage the multiple processors in today’s computers. Running threads on each processor simultaneously makes for faster applications.

Interprocess communication (IPC)

The means the OS provides to allow processes in the same application to talk to each other. Examples: sockets, message queuing.

This communication involves context switching at 3 points:

* Process switches from user mode (no privileges; programs run in this mode) to kernel mode (an elevated mode that gives access to all system resources)\*\*. This can be done, for example, via system call to open()\*.

\*http://stackoverflow.com/questions/11905934/how-to-switch-from-user-mode-to-kernel-mode

\*\* http://www.linfo.org/kernel\_mode.html

* In the kernel, the actual switching from one process to another takes place
* Switch from kernel mode to user mode

An application can also be composed of threads, but switching between threads is much cheaper since they communicate via their shared space, and the kernel need not be involved.

Thread Implementation

Methods to create threads, destroy threads, etc. can be packed into a library which runs in user mode and can be used by the application programmer to write multi-threaded programs. Or, the kernel can take ownership of the threads. The former approach is cheaper and faster when compared to the latter. It is cheaper because creating a thread merely entails allocating stack memory for the thread, and deleting it entails deallocating memory. It is faster because switching from one thread to another is not computationally expensive.

However, if a thread in a process makes a blocking call, the process is blocked as well all other threads in that process. Threads come in handy when they allow an application to perform several tasks at once. If a thread can cause other threads from running and performing tasks in parallel, then this is not very helpful. This can be remedied by moving their implementation to the kernel, but this is very expensive.

An answer to this problem is to use a mix of user-level threads and kernel-level threads called lightweight processes (LWPs). The OS allows user-level threads to still have their library of methods to create and destroy threads which can be used by multiple LWPs. The thread library is in the user space so the kernel is not involved, making this cheaper and faster. When a system call is made, thus creating an LWP (or several LWPs), the LWP(s) run(s) the scheduler in the user-level threads library, which looks for an available thread. The user space is responsible for making sure that multiple threads do not access to the thread table at the same time. When an available thread is found, the LWP context switches to it while other LWPs are also looking for available threads. Again, this takes place in the user space and the kernel is not involved.

When an LWP is blocked, there may be a context switch to another LWP, which will appear as normal code to the LWP since this is done in the user space. The other LWP will not continue from where it was earlier.

LWPs Advantages

*Cheap* – kernel not involved; thread handling done in user space

*No suspended process* – since the system switches over to available LWPs when one is blocked. This is assuming there are ample LWPs

*Transparent to users* – applications know nothing about the LWPs, it deals with the user-level threads

*Facilitates multiprocessing* – can run various LWPs on various CPUs

Multithreaded Clients

A multithreaded client gives the user a better experience because he/she is not while the client performs time consuming tasks (e.g. setting up TCP connection or waiting for a reply from a server far away). Multithreadedness allows the client to respond to the user on one thread while processing requests on another and setting up server connections on yet another, and with multiple threads, the client can connect to many servers at once. Of course, if the server is slow, then there is no performance gain. Some web servers are cloned and clients can set up connections to different clones so that data can be downloaded in parallel.

Multithreaded Servers

Server has a dispatcher thread that takes in client requests and forwards them to an available (worker) thread for processing. If the worker thread is suspended, another thread is chosen to run. If threads were not used, requests would be processed serially and take much longer. Also, the server will spend a lot of its time being idle, waiting for client requests to be completed.

Multi-threaded servers – Finite State Machine

After single threaded and multi-threaded server, you can opt for the server to be a finite state machine. Requests are handled on a lone, single thread. If the request can be fulfilled using data from the cache, it is processed. Otherwise it must go to the disk to fetch the data. Instead of blocking and waiting for a reply to this request from the disk, it records the state of the request in a table and looks at the next message. If it is a new request, it begins processing the request, if it is a reply from the disk, the information for the respective request is retrieved from the table and the request is processed and sent to client using a non-blocking call. Similarly, the server uses non-blocking system calls to receive messages as well. This makes it hard to program. Threads allow for the running of sequential processes and for *blocking* system calls, which are easier to program.

From Figure 3 -4 of Tanenbaum and Van Steen

Threads – Parallelism; blocking system calls (easier to program)

Single Thread – Serial execution; blocking calls (easy to progam)

Finite State Machine – Parallelism; non-blocking calls (hard to program)

3.2 - Virtualization

A thread can fool you into thinking there multiple CPUs, when in fact there is only one, but this concept can apply to other resources and is known as resource virtualization. It’s been around for a while but has experienced a revival since distributed systems have become the popular (and complicated), and the underlying system hardware and software are changing faster than the application software, i.e. you need to run older software on new system hardware and software.

Role of Virtualization in Distributed Systems

Virtualization allows for the running of legacy software on new platforms. This allows the platforms to be used for running a great many more programs since they are not just running new application software but can also run application software that has been around a while. Having a computer hooked up to a network means that a group of servers need to be maintained. Each server hosts programs that need to be remotely accessed by users in the network. If the applications each run on their own virtual machine (VM), on a common platform, there would not be need for as many physical machines, and the number of platforms to deal with will be less as well.

Architecture of Virtual Machines

As given in Figure 3-6 in Tanenbaum and Van Steen, virtualization mimics the following 4 interfaces in a computer:

Between HW and SW, consisting of machine instructions that can be invoked by any program

Between HW and SW, consisting of machine instructions that can only be invoked by privileged programs

OS has interface for system calls;

An interface of library calls that make up an API (that hides system calls)

2 different ways virtualization can occur:

As a process virtual machine: Build a system that gives abstract instruction set that may be interpreted (i.e. JRE) or emulated (cygwin). Virtualization is done for a single process.

As a layer (called a virtual machine monitor, VMM) that hides the original hardware but gives an interface that mimics the hardware’s capabilities and can be used by many different programs at once. This means you can different OSs running on the same platform. VMMs decouple hardware and software which aids portability. An example of such a layer is VMWare.

3.3 - Clients

Clients can communicate with remote servers in 2 ways:

* For each service that the client needs to access remotely, it will have the the same service locally that can contact the remote service over the network.
* Thin client approach (networked user interfaces) - Client is given an interface to the remote services. Client is dumb terminal; no local storage; application agnostic; everything done at server. More popular since more people have more and more apps on their mobile phones

3.3.1 - Networked User Interfaces

X Window System (X)

It is a networked user interface that controls bit-mapped terminals that consist of monitor, keyboard, and mouse.

X kernel has device drivers and also has low-level interface for manipulating the screen and handling mouse and keyboard events; Interface is usable by applications via library Xlib. The X kernel and X applications can be on different machines. X protocol is used to allow Xlib and X kernel to send data back and forth. I.e., X lib send request to X kernel to kill window, change the cursor, set colors, etc. The X kernel will handle keyboard and mouse events by sending event packets to Xlib

Window manager has special rights. It sets the look and feel of the windows as shown to the user. Once this theme is set, other applications must adhere to it.

X kernel fulfills requests from applications that have installed Xlib, and since the applications can be on other machines, this means that X kernel is acting as a server and applications are its clients.

Thin-Client Network Computing

Since client send requests over network to X kernel to be fulfilled, it would be make sense that the applications have the user interface and application logic decoupled. However, this is not the case which means that the application is waiting for responses in order to proceed. Bad for WANs due to high latencies.

Solution #1

Rewrite X protocol by compressing X messages to reduce bandwidth. Messages have fixed part (id) and variable part. Many messages with same id contain similar data. Better to send differences between messages with same id then sending all of them which would be redundant information.

On sender and receiver sides, entries are kept in cache and looked up via their ids. When message is sent, the cache is checked for the id. If the id is in the cache, then a message with this id was already sent but with possibly different data. Differential encoding is used to send only the differences in the data sent and that in the cache. When message is received, another cache lookup is performed and the differences are decoded. If an entry is not in the case, compression techniques are used. This has reduced bandwidth by factor of 1000.

Caching messages means that sending and receiving sides share information on current status of display. That is, an application can get information by looking it up in its local cache, which reduces number of messages to synchronize display and application.

X needs to have display server running with is a tall order on mobile phones. You could keep the display simple and put all processing on application and have the application completely control the display. This means that bitmap changes are sent over the network, from the application to the display. This requires advanced compression techniques, or bandwidth is a problem. Compression requires decompression at receiving end and that is computationally expensive unless you have expensive hardware.

Solution #2 -

THINC - high-level display commands that operate on device driver level; device dependent and more powerful than manipulating raw pixels, but less so than what X protocol has. Upshot is display servers simple (low CPU usage)

Think intercepts display requests and translates them to lower level commands. The translated commands are queued and sent in batches which means fewer messages. I.e., if one command overwrites another, then the first one will never go over the network. THINC pushes updates to display which saves latency since the display doesn’t have to send a request. This solution gives better performance.

Compound Documents - ???

In place editing - document wth text and graphics in word processor. When user places mouse on image, user interface sends information to drawing program to allow user to modify image. I.e., image is rotated and user interface sends new width and height of image to word processor

User interface that handles compound docs (a collection of possibly different kinds of documents), doesn’t let on that different applications are operating on different parts of it. Applications associated with compound doc do not have to run on client’s machine but keep in mind that the user interfaces that support compound docs do a lot more processing than those that don’t

3.3.2 - Client Side Software for Distribution Transparency

Distribution transparency - user interface are used to get distribution transparency

Access transparency - handled via client stub that offers same functionality as at the server but makes the differences in architecture invisible to the user

Location, migration relocation transparency - client’s middleware can hide server’s location and bind to the server without the user’s knowledge. Worst case scenario, application experiences temporary drop in performance

Relocation transparency - if there are replicated servers in a distributed system, the client software can send a request to each copy, gather all responses, and send one response back to the client application

Failure transparency - client middleware can repeated attempt to connect or try another server after failing a certain number of times. Or, it can return cached data if connection fails.

Concurrency Persistence transparency - handled at server

Servers

3.4.1 General Design Issues

A server is a process that provides a service to clients who connect to it. It listens on an endpoint (port) for a request from a client and when it gets one, it processes it and waits for the next request.

Iterative server - no threads, server does work and sends response to client

Concurrent - server gives task to create multiple threads (or fork processes) to complete and continues waiting. Unix forks process that get back to the client with a response.

Server has assigned some endpoints (ports) assigned by Internet Assigned Number Authority (IANA) . For example, TCP = port 21, HTTP = port 80. Since ports are known, clients only need IPs of machines to use these services.

Sometimes, ports area assigned dynamically at the server so the client will have to look up this information. A server could have a background process running that knows (via an Endpoint table) which endpoints currently belong to which service on a co-located server. Endpoint of background process (daemon) is known so client just asks it for the endpoint and then contact that server.

If you have each service on one server, this is a waste. Unix has many servers (processes) that run at the same time, waiting for a request, which means that there are a lot of idle processes to maintain. A better approach is to use a superserver (a daemon process) that listens at each port with a given service running on it. intend daemon does things this way. It listens on ports and when a request comes in, a process is forked to handle it and the process will exit when it is finished.

Interrupting a server. What to do when a user cancels download of file in the middle of the download?

* (Silly) User can restart application to break connection and reconnect
* (Better) Have client and server be able to send out-of-band data, i.e., data to be processed before any other data from that client. Server listens to control endpoint for out-of-band data from client and also listens (with lower priority) to port for normal data (page 90).

Or, send out-of-band data in same connection as client’s original request. When TCP sends urgent data to the server, the original request is interrupted and the urgent data is examined and processed

Statelessness - a server does not keep track of what data was given to whom, it just responds to requests as they come in and forgets the client when the connection is closed. In some designs, stateless servers do keep track but if this information is lost, this is not a showstopper. Web servers logs client requests but if this is lost, there is no harm done since this just to see which documents are getting uploaded most. The server can stil continue to provide pages without this information.

soft state - state is only maintained for a while, after that it is discarded. Example: server that notifies client of updates for a while but has to be polled for updates by the client after that.

Statefullness - keeps information on clients from one session to another. Stateful servers have better performance that stateless on the minus side, if the server crashes then the whole state has be recovered so that it is as it was before the server crashed.

session state vs permanent state

session state uses 3-tiered client-server architecture (user, processing, and data levels). Application gets data for client from database and if session state is lost then there is no harm done if the client can resend the query.

permanent state - covers data \*in\* the databases

stateless or stateful should not figure into the services the server provides. That is, a stateless server should figure out a way to open a file before it can be read. Or, a server keeps track of client s past behavior to that it can make recommendations in the future. If stateless, server can go the cookie route. That is, send a cookie along with web pages and whenever server is contacted, the server’s cookie is sent too.

3.4.2 - Server Clusters (LAN)

A collection of connected machines, where each machine is running one or more processes

General Organization - 3 tiers

logical switch - routes requests from clients. In case of transport layer, they are routed to a process in the cluster. In the case of HTTP, requests may be sent to application servers

application processing - processes run on fast hardware, or on low end machines when tasks are not computationally intensive is not the cause of logjams, but access to memory is

database-processing - i.e. database servers, which could be on high performance machines

This 3-tiered organization is not set in stone. Can have machines with their own memory storage so application and data processing in one process, leading to 2-tier structure

Could have different machines running different application processes if the cluster has more than on service. This means that the switch tier will have to tell one process from another so the requests are routed properly.

Want to have access transparency so server cluster has only one access point.

Can send requests to server over TCP in a session, when request is completed, session is torn down. For transport layer switches, a TCP connection request is accepted, a process is found to handle the requests, and the request is sent to that process (TCP handoff). The process sends ACK to client, adding switch’s IP as source since that is who the client is communicating with (and thus, that is who the client is expecting a reply from).

Switch decides where to send request so that means it also chooses the process to handle it. So as not to overload any one process, switch makes this choice round-robin style, each time, picking the next process on its list. Or, could use more advanced selection techniques. Can have transport layer make process selection if processes have different port numbers. Or, look at payload, if you know what to expect in the payload.

Distributed Servers

Machines and access points in distributed clusters change on the fly, but appear to be one stable unit

This is achieved using mobile support for IPv6 (MIPv6). Basically, a mobile node lives in a home network and has a home address (HoA) in this network which is stable. The home network contains a home agent that routes traffic to that mobile when it leaves home. When the mobile enters a different network, it gets a temporary care-of-address (CoA) that is sends to its home agent so the home agent can reach it.

This can be used to allow a distributed server to have a stable address. The cluster is given one, unique, stable contact address that is used for the lifetime of the server and is used to talk to the outside. Only one node at any given time acts as the access point under the contact address, but this task can be passed of to another node, wherein the new node will then give \*its\* home address as the care-of-address. The access point gives its home address as a care-of-address to the home agent and hereafter, all traffic will go to the access point, who will forward requests to active nodes.

Since this means that home agent and access points are logjams, use route optimization, a feature of MIPv6. The client uses a home address to communicate with the distributed server. When a mobile of home address HA reports its care-of-address (CA) to the home agent, the home agent can forward that information to client which stores the tuple (HA,CA). After this point, communication is sent to CA directly even if the client application uses a home address, this is converted CA.

Route optimization can make multiple clients think that they are talking to the same process, when in fact, they are talking to different nodes in the system. For example, client C1’s request is sent to home address HA. The access point sends the request to node S1 with care of address CA1. S1 then tells C1 that the care of address is CA so client stores the tuple (HA, CA1).

In the meantime, C2 sends a request to HA. This is forwarded to node S2 with care-of-address CA2 and C2 stores the tuple (HA, CA2). CA1 and CA2 are 2 different machines in the distributed server but each thinks the machine address is HA.

3.4.3 - Managing Server Clusters

Common Approaches

One way to manage server clusters is to have an administrator log in remotely to a machine in the cluster and install or change components, monitor the machines, etc.

A more modern approach would be to have an interface on an administrator machine that hides the fact that you need to log into a node and allows administrators to work on multiple machines. I.e., Cluster Systems Management by IBM.

However, once the cluster gets to several tens of nodes, there is no real answer.

Example: PlanetLab

Machines are donated by companies. Each machine has an enchanted Linux OS, Virtual Machine Monitor (VMM), that supports vservers, which are separate environments in which to run groups of processes. The processes in different vservers have absolutely nothing to do with one another and do not share resources. This means that one vserver can be running Python 2 and other can be running Python 3. The VMM makes sure that the vservers are isolated from one another.

A slice is a collection of vservers from different nodes, i.e., a VM cluster.

node manager - responsible for creating vservers and managing resource allocation. An rspec gives the time interval when resources have been allocated, and is referenced via an rcap, a 128-bit id. The node manager uses the rcap to find the rspec in a table saved locally on the node.

service provider - need slice to use resources and each slice is paired with something (a business, school, etc.) that has an account on PlanetLab.

slice creation service (SCS) - creates a slice; is run on each node. SCS contacts node manager for a vserver and resource allocation. Node manager not contacted directly and not just anybody can ask the SCS to create a slice.

Only slice authorities can request slice creation and they have credentials to a collection of nodes. Service providers contacts them to create a slice across nodes. Service provider and slice authority have a prior working relationship.

management authorities - make sure that nodes under its purview run PlanetLab software and are in compliance with PlanetLab rules

From page 102, the relationships are as follows:

1. Node owner puts node under purview of management authority
2. Management authority provides software to add to PlanetLab
3. Service provider registers with management authority
4. Service provider contacts slice authority to create slice on collection of nodes
5. Slice authority need to authenticate service provider
6. Node owner provides SCS for slice authority
7. Management authority delegates creation of slices to slice authority

To monitor users’ programs, each node has sensors that report CPU usage, disk activity, etc